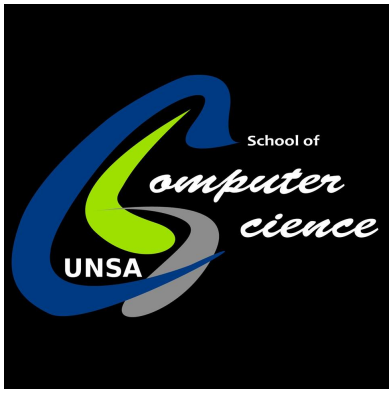
UNIVERSIDAD NACIONAL
SAN AGUSTIN



Universidad Nacional de San Agustín

Escuela Profesional de Ciencia de la Computación

Informe de algoritmos de ordenamiento
Alumno: Villanueva Flores Luis Guillermo
Docente: Vicente Machaca Arceda
Curso: Estructuras de datos avanzada

2019

Informe de Algoritmos de Ordenamiento

1. **Algoritmo: Bubble Sort**

   Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas burbujas. También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo el más sencillo de implementar.

   a) Codigo fuente en C++

```
void bubble(int *A, int n) {
  int x,y,tmp;
  for(x = 1; x < n; x++) {
    for(y = 0; y < n - x; y++) {
      if(A[y] > A[y + 1]) {
        tmp = A[y];
        A[y] = A[y + 1];
        A[y + 1] = tmp;
      }
    }
  }
}
```

   Pasamos como parametro el arreglo y el tamaño del arreglo, y luego hacemos la comparacion de los elementos, para ello usamos un doble “for”, para ir comparando cada elemento con todos los elementos, debido a este doble bucle el tiempo es n^2

b) Código fuente en Java

```java
class BubbleSort
{
    void bubbleSort(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n-1; i++)
            for (int j = 0; j < n-i-1; j++)
                if (arr[j] > arr[j+1])
                {
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
    }
```

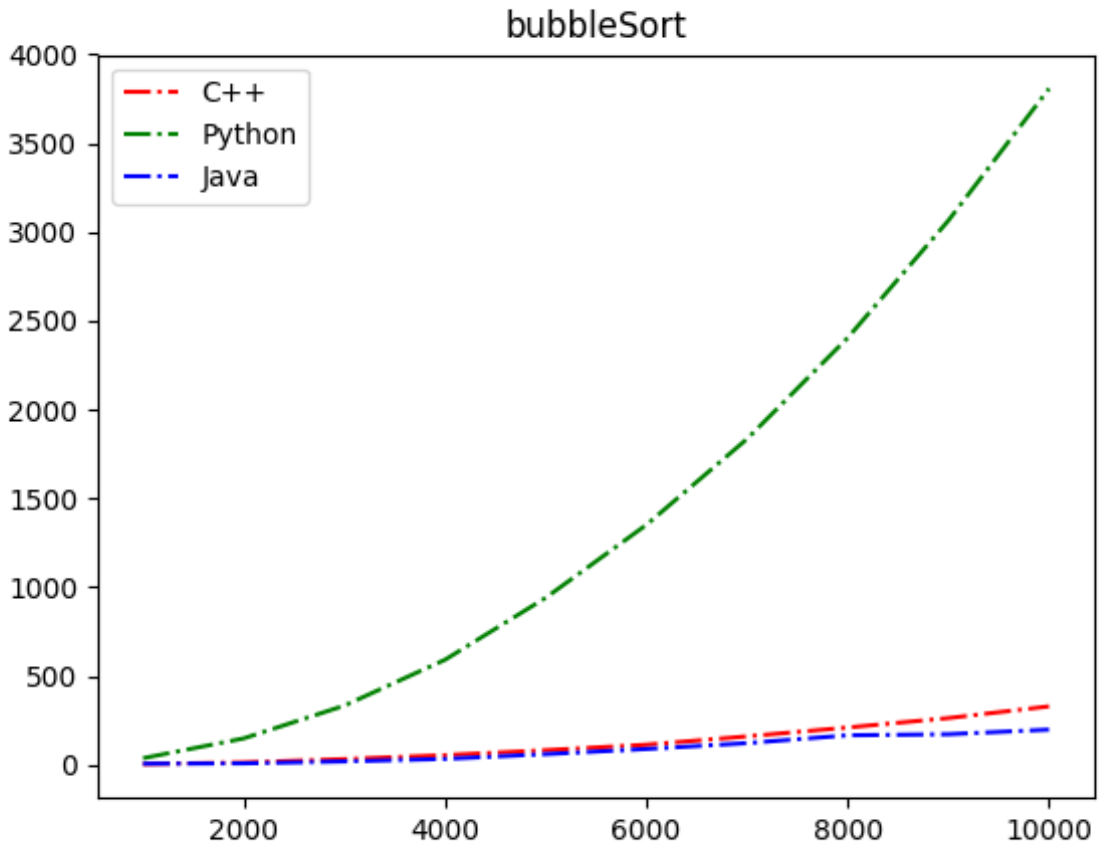
Al igual que en C++ usamos doble "for"para el algoritmo pero por parámetro de función pasamos simplemente el arreglo y dentro de la función calculamos su tamaño para poder trabajar.

c) Código fuente en Python

```python
def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Al igual que en C++ y Java usamos doble "for"para el algoritmo pero por parámetro de función pasamos simplemente el arreglo y dentro de la función calculamos su tamaño para poder trabajar.

d) Gráfica comparativa del Bubble Sort



Como observamos para el caso del lenguaje python es muy costoso, debido a que los bucles en python de por sí son costosos y al usar uno doble con mayor razón, por otro lado vemos que Java y C++ están a la par, teniendo a Java con una cierta ventaja debido a que en estos lenguajes los bucles no son tan costosos como en el anterior.

2. **Algoritmo: Insertion Sort**

Es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria. Requiere O(n^2) operaciones para ordenar una lista de n elementos.

Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay k elementos ordenados de menor a mayor, se toma el elemento k+1 y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se inserta el elemento k+1 debiendo desplazarse los demás elementos.

a) Codigo fuente en C++

```
void insertsort(int *A, int n) {
  int x,val,y;
  for(x = 1; x < n; x++) {
    val = A[x];
    y = x - 1;
    while (y >= 0 && A[y] > val) {
      A[y + 1] = A[y];
      y--;
    }
    A[y + 1] = val;
  }
}
```

Pasamos como parametro el arreglo y el tamaño del arreglo, y luego hacemos la comparacion de los elementos, para ello usamos un "for"que comienza en a pocisión 1 del arreglo y luego usamos un "while"para ir haciendo la comparación con todos los elementos para encontrar la posición indicada.Debido a este doble bucle el tiempo es n^2 .

b) Código fuente en Java

```java
class InsertionSort {
    void sort(int arr[])
    {
        int n = arr.length;
        for (int i = 1; i < n; ++i) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }
```

Al igual que en C++ usamos un "forz un "while"para el algoritmo, pero por parámetro de función pasamos simplemente el arreglo y dentro de la función calculamos su tamaño para poder trabajar.Todo el procedimiento es igual en todos los lenguajes.

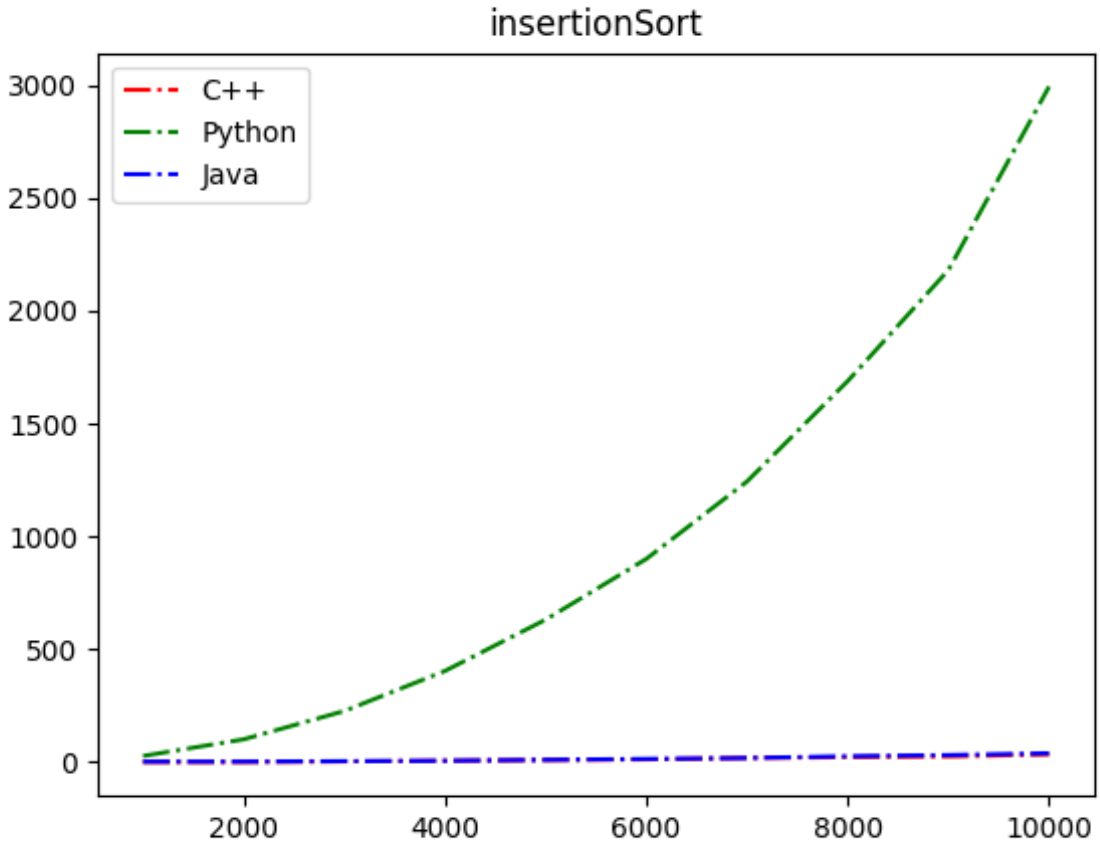
c) Código fuente en Python

```python
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key
```

Al igual que en C++ y Java usamos un "forz un "while"para el algoritmo, pero por parámetro de función pasamos simplemente el arreglo y en el primer bucle vamos hasta el tamaño del mismo, y seleccionamos nuestra llave que comenzará en 1 y es el elemento que se irá comparando con el anterior e

irá adelante de ser menor, de lo contrario se mantendrá en su posición y el siguiente elemento repetirá esto.

d) Gráfica comparativa del Insertion Sort



Al igual que en el bubble al ser tiempo n^2 el tiempo sería muy similar y para el caso del lenguaje python es muy costoso, debido a que los bucles en python de por sí son costosos y al usar uno doble con mayor razón, por otro lado vemos que Java y C++ están a la par, por no decir iguales.

3. **Algoritmo: Selection Sort**

Consiste en encontrar el menor de todos los elementos del arreglo o vector e intercambiarlo con el que está en la primera posición. Luego el segundo mas pequeño, y así sucesivamente hasta ordenarlo todo. Su implementación requiere O(n^2) comparaciones e intercambios para ordenar una secuencia de elementos. Este algoritmo mejora ligeramente el algoritmo de la burbuja. En el caso de tener que ordenar un vector de enteros,

esta mejora no es muy sustancial, pero cuando hay que ordenar un vector de estructuras más complejas, la operación de intercambiar los elementos sería más costosa en este caso.

a) Codigo fuente en C++

```
void selectionsort(int *A, int n) {
  int x, y, min, tmp;
  for(x = 0; x < n; x++) {
    min = x;
    for(y = x + 1; y < n; y++) {
      if(A[min] > A[y]) {
        min = y;
      }
    }
    tmp = A[min];
    A[min] = A[x];
    A[x] = tmp;
  }
}
```

Pasamos como parametro el arreglo y el tamaño del arreglo, y luego hacemos la comparacion de los elementos, para ello usamos doble "for"que comienza en a pocisión 0 del arreglo y va buscando un nuevo menor en cada iteración y lo va reemplazando, o sea hace una especia de "swap".Debido a este doble bucle el tiempo es n^2 .

b) Código fuente en Java

```java
class SelectionSort
{
    void sort(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n-1; i++)
        {
            int min_idx = i;
            for (int j = i+1; j < n; j++)
                if (arr[j] < arr[min_idx])
                    min_idx = j;
            int temp = arr[min_idx];
            arr[min_idx] = arr[i];
            arr[i] = temp;
        }
    }
```

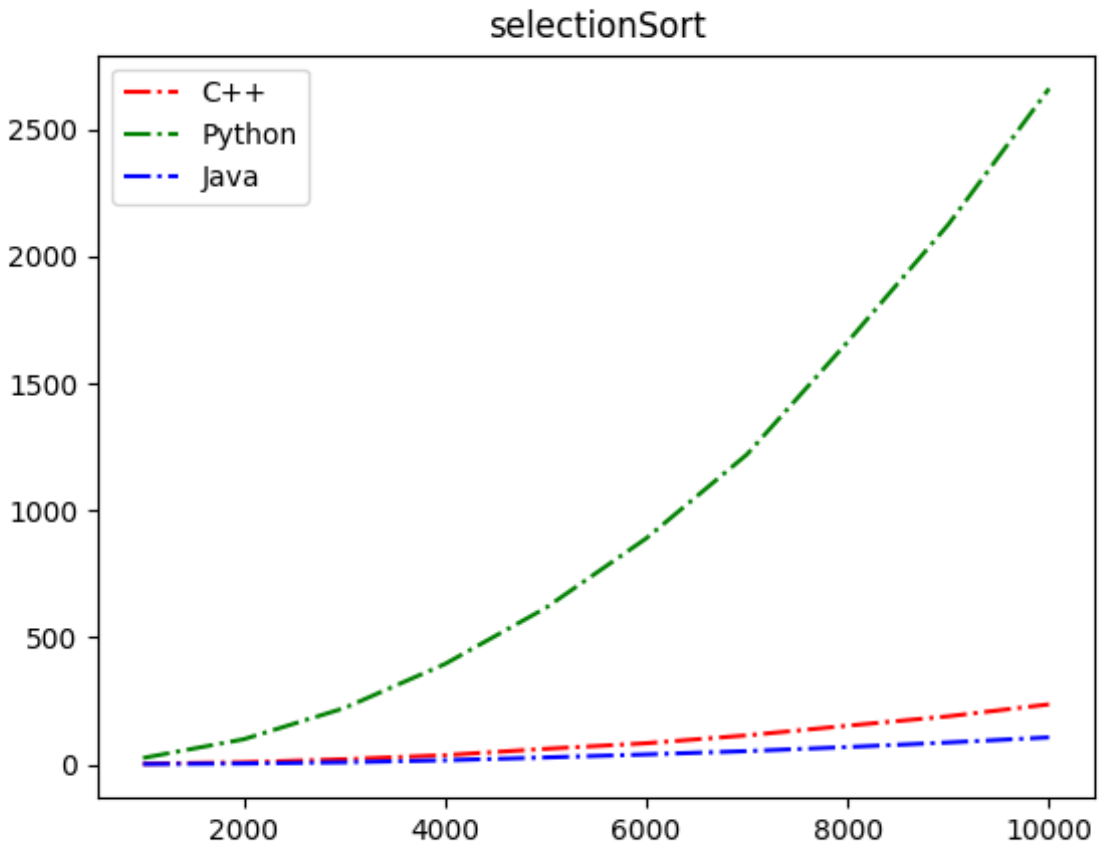
Al igual que en C++ usamos doble "for"para recorrer el arreglo y buscar el menor .Todo el procedimiento es igual en todos los lenguajes.

c) Código fuente en Python

```python
def selectionSort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[min_idx] > arr[j]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

Al igual que en C++ y Java usamos doble "for"para recorrer el arreglo y buscar el menor sólo que en la parte de hacer el cambio de valores el lenguaje nos permite hacerlo de la manera que muestra la imagen .Todo el procedimiento es igual en todos los lenguajes.

d) Gráfica comparativa del Selection Sort



Al igual que en el bubble y el insert al ser tiempo n^2 valga la redundancia el tiempo es similar pero este algoritmo es menos costoso que los otros anteriores mencioandos pero no deja de ser n^2.Para el caso del lenguaje python es muy costoso, debido a que los bucles en python de por sí son costosos y al usar uno doble con mayor razón, por otro lado vemos que Java y C++ están a la par con una ventaja de Java pero no de gran diferencia, ya que ambos lenguajes son parecidos, se da este tipo de gráficas.

4. **Algoritmo: Counting Sort**

Es un algoritmo de ordenamiento en el que se cuenta el número de elementos de cada clase para luego ordenarlos. Sólo puede ser utilizado por tanto para ordenar elementos que sean contables (como los números enteros en un determinado intervalo, pero no los números reales, por ejemplo).
El primer paso consiste en averiguar cuál es el intervalo dentro del que están los datos a ordenar (valores mínimo y máximo). Después se crea un vector de números enteros con tantos elementos como valores haya

en el intervalo [mínimo, máximo], y a cada elemento se le da el valor 0 (0 apariciones). Tras esto se recorren todos los elementos a ordenar y se cuenta el número de apariciones de cada elemento (usando el vector que hemos creado). Por último, basta con recorrer este vector para tener todos los elementos ordenados.
Se trata de un algoritmo estable cuya complejidad computacional es O(n+k), siendo n el número de elementos a ordenar y k el tamaño del vector auxiliar (máximo - mínimo). La eficiencia del algoritmo es independiente de lo casi ordenado que estuviera anteriormente. Es decir no existe un mejor y peor caso, todos los casos se tratan iguales. El algoritmo counting, no se ordena in situ, sino que requiere de una memoria adicional.

a) Codigo fuente en C++

```
int getMax(int array[], int size) {
   int max = array[1];
   for(int i = 2; i<=size; i++) {
      if(array[i] > max)
         max = array[i];
   }
   return max;
}
void countSort(int *array, int size) {
   int output[size+1];
   int max = getMax(array, size);
   int count[max+1];
      count[i] = 0;
   for(int i = 1; i <=size; i++)
      count[array[i]]++;
   for(int i = 1; i<=max; i++)
      count[i] += count[i-1];
   for(int i = size; i>=1; i--) {
      output[count[array[i]]] = array[i];
      count[array[i]] -= 1;
   }
   for(int i = 1; i<=size; i++) {
      array[i] = output[i];
   }
}
```

Pasamos como parametro el arreglo y el tamaño del arreglo, también tenemos una función para calcular el mayor, ya que esa nos servira al momento de crear el arreglo auxiliar que es el que va a contar, siguiendo la teoría del algoritmo realizamos todos lo pasos, primero inicializando el array que contador en cero, luego le sumamos en la posición del arreglo que cuenta el número que hay en el arreglo original,

luego hacemos una suma acumulada y por último hacemos la operación del algoritmo que significaría reemplazar en el arreglo de salida el número que hay en el arreglo inicial y buscar ese número en la posición del arreglo contador y dependiendo el número que contenga dicho arreglo se insertará en esa posición el arreglo de salida. Esa sería la explicación del Counting, la misma que funciona para C++, Java y Python.

b) Código fuente en Java

```java
class CountingSort {
    void sort(char arr[])
    {
        int n = arr.length;
        char output[] = new char[n];
        int count[] = new int[256];
        for (int i = 0; i < 256; ++i)
            count[i] = 0;

        for (int i = 0; i < n; ++i)
            ++count[arr[i]];
        for (int i = 1; i <= 255; ++i)
            count[i] += count[i - 1];

        for (int i = 0; i < n; ++i) {
            output[count[arr[i]] - 1] = arr[i];
            --count[arr[i]];
        }
        for (int i = 0; i < n; ++i)
            arr[i] = output[i];
    }
}
```

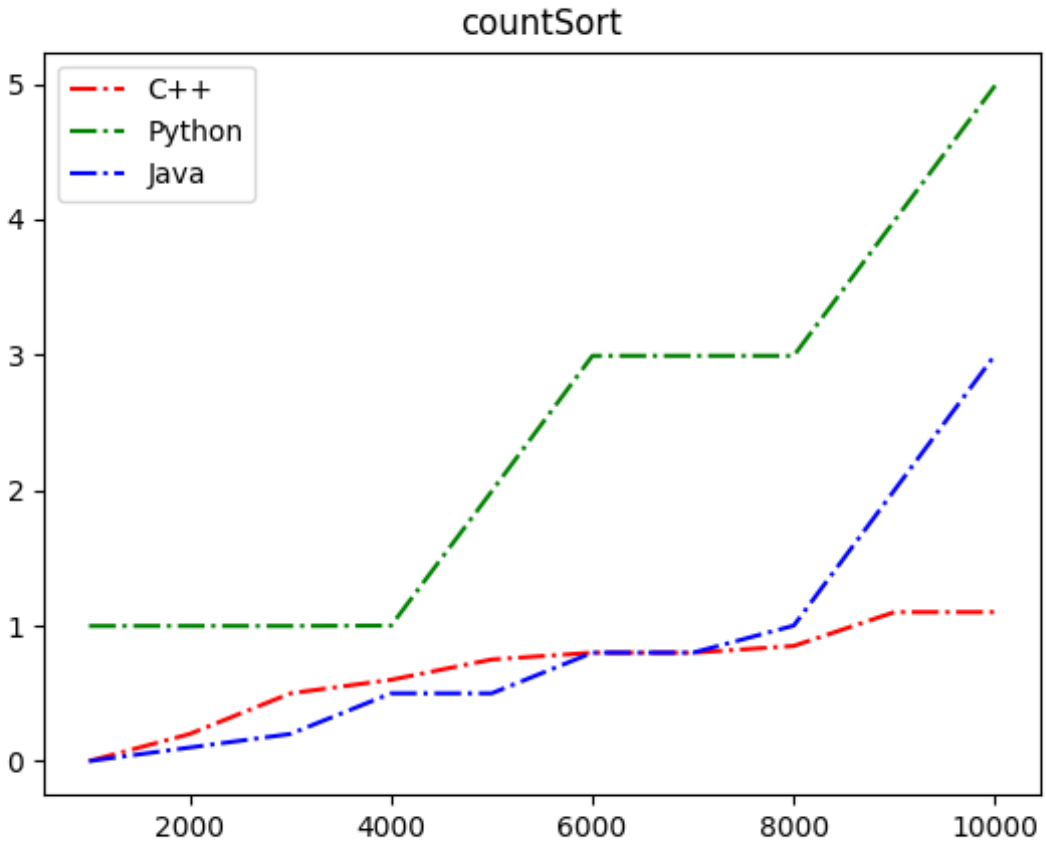
La lógica del algoritmo es la misma, sólo que para este lenguaje usamos un arreglo de caracteres debido a que ordenamermos caracteres, esto no debería ser ningún inconveniente ya que el algoritmo también esta adaptado para ordenar caracteres.

c) Código fuente en Python

```python
def countSort(arr):
    output = [0 for i in range(256)]
    count = [0 for i in range(256)]
    ans = ["" for _ in arr]
    for i in arr:
        count[i] += 1
    for i in range(256):
        count[i] += count[i-1]
    for i in range(len(arr)):
        output[count[arr[i]]-1] = arr[i]
        count[arr[i]] -= 1
    for i in range(len(arr)):
        ans[i] = output[i]
```

Al igual que en C++ y Java la lógica del algoritmo es la misma, sólo que aqui al igual que en Java solo pasamos el arreglo como parámetro y el rango que le damos por defecto va a ser de 256, por todo lo demás la lógica del prrograma se mantiene.

d) Gráfica comparativa del Counting Sort



Al ser un algoritmo no tan costoso, para ser exactos su costo es de O(n+k), no demora mucho en ejecutarse y los tiempos que bota son pequeños para un arreglo de 10000 elementos, aún así esa sería la constante del tiempo del Counting, como siempre Python es el más costoso debido a sus bucles, mientras que C++ y Java se mantienen a la par, pero sacando una ventaja C++.

5. **Algoritmo: Heap Sort**

Es un algoritmo de ordenamiento no recursivo, no estable, con complejidad computacional O(n log n) .
Este algoritmo consiste en almacenar todos los elementos del vector a ordenar en un montículo (heap), y luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados en él. El algoritmo, después de cada extracción, recoloca en el nodo raíz o cima, la última hoja por la derecha del último nivel. Lo cual destruye la propiedad heap del árbol. Pero, a continuación realiza un proceso de "descenso"del número insertado de forma que se elige a cada movimiento el mayor de sus dos hijos, con el que se intercambia. Este intercambio, realizado sucesivamente "hunde.[el] nodo en el árbol restaurando la propiedad montículo del árbol y dejando paso a la siguiente extracción del nodo raíz.

El algoritmo, en su implementación habitual, tiene dos fases. Primero una fase de construcción de un montículo a partir del conjunto de elementos de entrada, y después, una fase de extracción sucesiva de la cima del montículo. El vector definitivo y ordenado, empieza a construirse por el final y termina por el principio.

a) Codigo fuente en C++

```
void heapify(int arr[], int n, int i)
{
    int largest = i;
    int l = 2*i + 1;
    int r = 2*i + 2;

    if (l < n && arr[l] > arr[largest])
        largest = l;
    if (r < n && arr[r] > arr[largest])
        largest = r;
    if (largest != i)
    {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
void heapSort(int arr[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i=n-1; i>=0; i--)
    {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

Tenemos dos funciones, una que se encargara de crear el montículo (o heap en ingles) y la otra que se encargará del ordenamiento en sí pero que llama a la función del montículo.

b) Código fuente en Java

```java
public class HeapSort
{
    public void sort(int arr[])
    {
        int n = arr.length;
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);
        for (int i=n-1; i>=0; i--)
        {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
            heapify(arr, i, 0);
        }
    }
    void heapify(int arr[], int n, int i)
    {
        int largest = i;
        int l = 2*i + 1;
        int r = 2*i + 2;
        if (l < n && arr[l] > arr[largest])
            largest = l;

        if (r < n && arr[r] > arr[largest])
            largest = r;

        if (largest != i)
        {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;
            heapify(arr, n, largest);
        }
    }
```

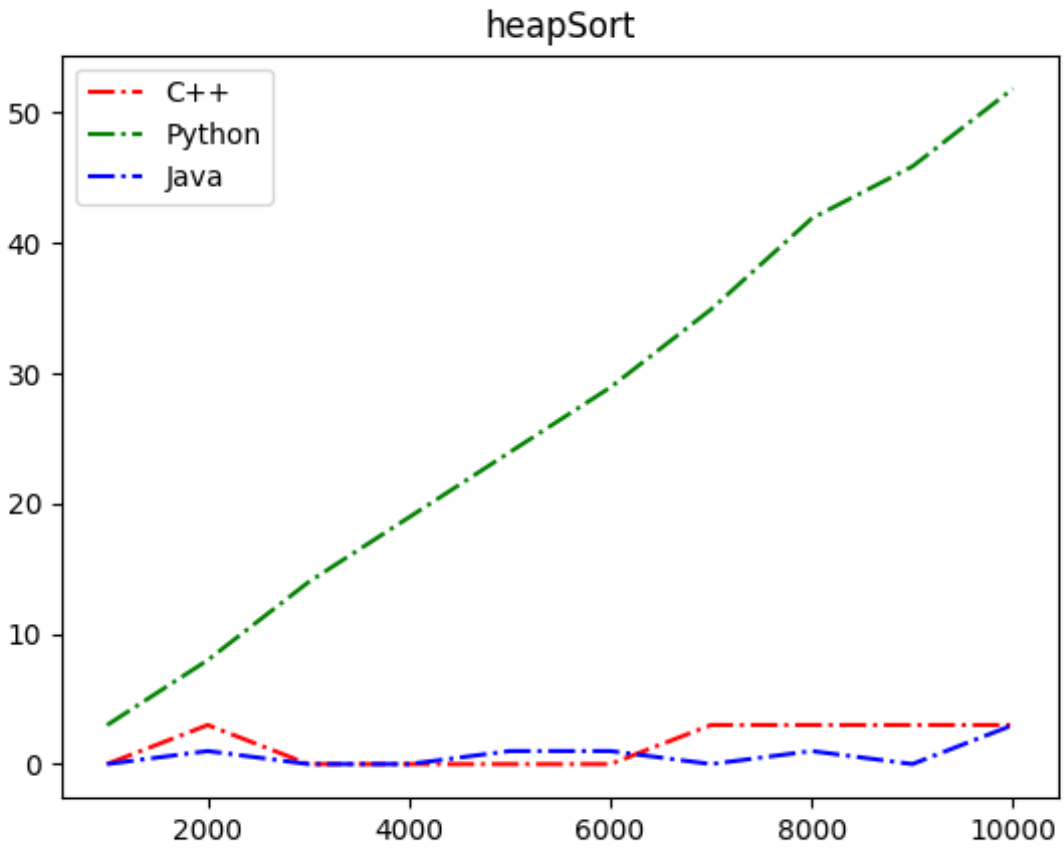
Al igual que en C++ contamos con dos funciones, una encargada del montículo que de por sí ya va haciendo el ordenamiento y otra que va a ir llamando a la función del montículo según lo requiera para ordenar el arreglo.

c) Código fuente en Python

```python
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[i] < arr[l]:
        largest = l
    if r < n and arr[largest] < arr[r]:
        largest = r
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i]
        heapify(arr, n, largest)
def heapSort(arr):
    n = len(arr)
    for i in range(n, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

Al igual que en C++ y Java la lógica del algoritmo es la misma.

d) Gráfica comparativa del Heap Sort



Al ser un algoritmo con complejidad O(n log n) no es tan costoso y lo podemos verificar al observar la gráfica y corroborar con los tiempos que se demora para cada lenguaje, siendo Python siempre el que más se demora y como constante Java y C++ se mantienen a la par con alguna ventaja mínima de Java sobre C++.

6. **Algoritmo: Quick Sort**

QuickSort (en inglés, ordenamiento rápido). Es un algoritmo basado en la técnica de divide y vencerás, que permite, en promedio, ordenar n elementos en un tiempo proporcional a n log n.
El algoritmo consta de los siguientes pasos:

a) Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote.

b) Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.

c) La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.

d) Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento.

A continuación se muestran las imagenes del código fuente en cada lenguaje:

a) Codigo fuente en C++

```
void Quicksort(int *A, int inicio, int final) {
  int i = inicio, f = final, tmp;
  int x = A[(inicio + final) / 2];
  do {
    while(A[i] < x && f <= final) {
      i++;
    }
    while(x < A[f] && f > inicio) {
      f--;
    }
    if(i <= f) {
      tmp = A[i];
      A[i] = A[f];
      A[f] = tmp;
      i++; f--;
    }
  } while(i <= f);

  if(inicio < f) {
    qsort(A,inicio,f);
  }

  if(i < final){
    qsort(A,i,final);
  }

}
```

En esta función hacemos la eleccion del pivote dentro de la función Quicksort y la partición también se

hace dentro, con el hecho de no crear otra función aparte, la elección del pivote ha sido considerando el elemento del medio, la recursividad funciona de la misma manera que se explica el algoritmo.

b) Código fuente en Java

```java
class QuickSort
{
    int partition(int arr[], int low, int high)
    {
        int pivot = arr[high];
        int i = (low-1);
        for (int j=low; j<high; j++)
        {
            if (arr[j] <= pivot)
            {
                i++;
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        int temp = arr[i+1];
        arr[i+1] = arr[high];
        arr[high] = temp;

        return i+1;
    }

    void sort(int arr[], int low, int high)
    {
        if (low < high)
        {
            int pi = partition(arr, low, high);
            sort(arr, low, pi-1);
            sort(arr, pi+1, high);
        }
    }
```

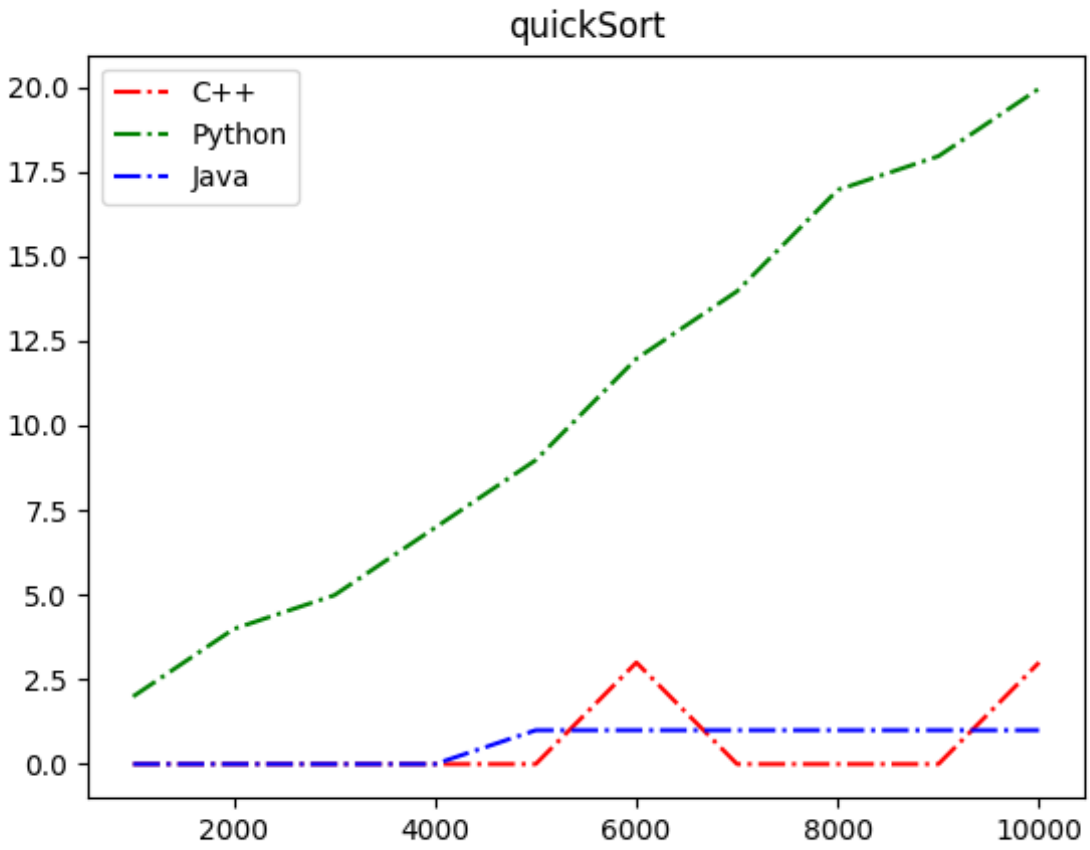
A diferencia de C++ contamos con dos funciones, una encargada de la particion y otra encargada de llamar a la partición si en caso lo requiera, como ya vimos en la especificación del algoritmo.Cabe realtar que la función partición devuelve un entero que será en que intervalos del arreglo original se tomara para hacer el ordenamiento, uno ya sea para la izquierda del pivote y otro para su derecha.Para este caso el pívote será el último elemento del arreglo.

c) Código fuente en Python

```
def partition(arr,low,high):
    i = ( low-1 )
    pivot = arr[high]
    for j in range(low , high):
        if   arr[j] < pivot:
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]
    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )
def quickSort(arr,low,high):
    if low < high:
        pi = partition(arr,low,high)
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)
```

Al igual que en Java la lógica del algoritmo es la misma también para la elección del pivote.

d) Gráfica comparativa del Quick Sort



Al ser un algoritmo con complejidad O(n log n) no es tan costoso y lo podemos verificar al observar la gráfica y corroborar con los tiempos que se demora para cada lenguaje, siendo Python siempre el que más se demora y como constante Java y C++ se mantienen a la par con alguna ventaja mínima de Java sobre C++. Vemos que para el caso de C++ en el tamaño 6000 hay una especie de declinación, esto se debe a que al momento de generar los datos a un archivo .csv no salen todos de la forma entera o decimal exacta , por ello que aveces el programa los redondea y se produce esos escalones, pero manteniendose constante durante toda la grafica después.
Vemos que ese problema ocurre para C++ y sí porque los tiempos generados en C++ no son muy exactos y nos ocurre ese inconveniente, pero para Java si se mantiene estable.

7. **Algoritmo: Merge Sort**

El algoritmo de ordenamiento por mezcla (merge sort en inglés) es un algoritmo de ordenamiento externo estable basado en la técnica divide y vencerás. Es de complejidad O(n log n).
Sigue los siguientes pasos:

a) Si la longitud de la lista es 0 ó 1, entonces ya está ordenada. En otro caso:
b) Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
c) Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
d) Mezclar las dos sublistas en una sola lista ordenada.

El algoritmo incorpora dos ideas principales para mejorar su tiempo de ejecución:

a) Una lista pequeña necesitará menos pasos para ordenarse que una lista grande.
b) Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que a partir de dos listas desordenadas.

A continuación se muestra el código fuente del Merge para los diferentes lenguajes:

a) Codigo fuente en C++

```
void merges(int arr[], int l, int m, int r){
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;
    int L[n1], R[n2];
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    i = 0;j = 0;k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j]){
            arr[k] = L[i];
            i++;
        }
        else{
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1){
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2){
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Esta sería la función que se encargaría de separar el arreglo original dependiendo los límtes que se le pasen, y esta también es la que se encargaría de hacer la mezcla. Como podemos ver por parámetro se le pasa el arreglo, el limite izquierdo,derecho y la posición de la mitad que serán necesarios para que de una forma implícita separe el arreglo en dos y pueda hacer la mezcla.

```
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = l+(r-l)/2;
        mergeSort(arr, l, m);
        mergeSort(arr, m+1, r);
        merges(arr, l, m, r);
    }
}
```

Esta función es la que se encargará de llamar recursivamente a la función mezlca y separar la lista hasta que haya solo 1 elemento para que al momento que regrese en la recusrividad se encargue de hacer la mezcla.

b) Código fuente en Java

```
class MergeSort {
    void merge(int arr[], int l, int m, int r) {
        int n1 = m - l + 1;
        int n2 = r - m;
        int L[] = new int [n1];
        int R[] = new int [n2];
        for (int i=0; i<n1; ++i) {L[i] = arr[l + i];}
        for (int j=0; j<n2; ++j) {R[j] = arr[m + 1+ j];}
        int i = 0, j = 0;
        int k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            }
            else{
                arr[k] = R[j];
                j++;
            }
            k++;
        }
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }
        while (j < n2) {
            arr[k] = R[j];
            j++;
            k++;
        }
    }
```

Al igual que en C++ contamos con dos funciones,la primera que se muestra a continuación que se encargaría al igual que en C++ de hacer la mezcla.

```
void sort(int arr[], int l, int r)
{
    if (l < r)
    {
        int m = (l+r)/2;
        sort(arr, l, m);
        sort(arr , m+1, r);
        merge(arr, l, m, r);
    }
}
```

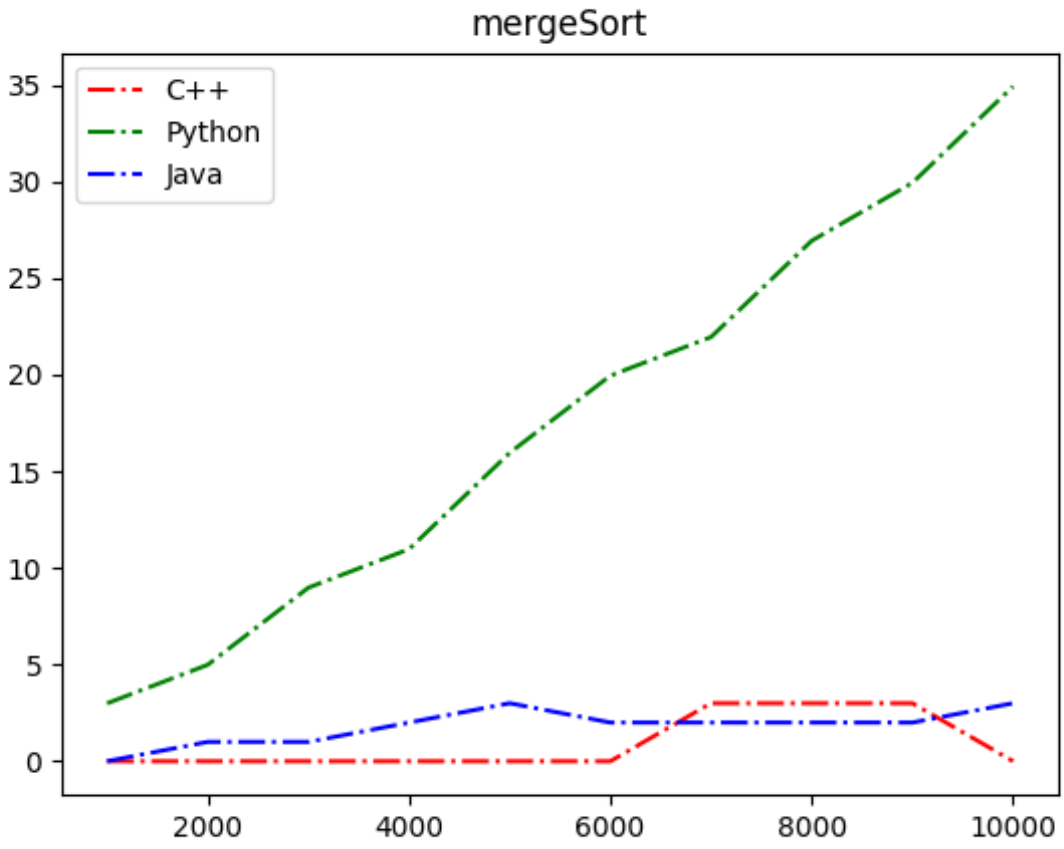
Y esta función mergesort que se encarga de separar el arreglo en dos límites hasta que el número de elementos sea igual a 1, para de ahí recién poder hacer la mmezcla.

c) Código fuente en Python

```
def mergeSort(arr):
    if len(arr) >1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L)
        mergeSort(R)
        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i+=1
            else:
                arr[k] = R[j]
                j+=1
            k+=1
        while i < len(L):
            arr[k] = L[i]
            i+=1
            k+=1
        while j < len(R):
            arr[k] = R[j]
            j+=1
            k+=1
```

Al igual que en C++ y Java la lógica del algoritmo es la misma, sólo que en este caso no se usan dos funciones, sino una sola pero se hace la verificación en la primera parte del código para separar la lista hasta que sea uno y después se hace la operación de la mezcla, sigue la misma lógica que en los dos anteriores lenguajes.

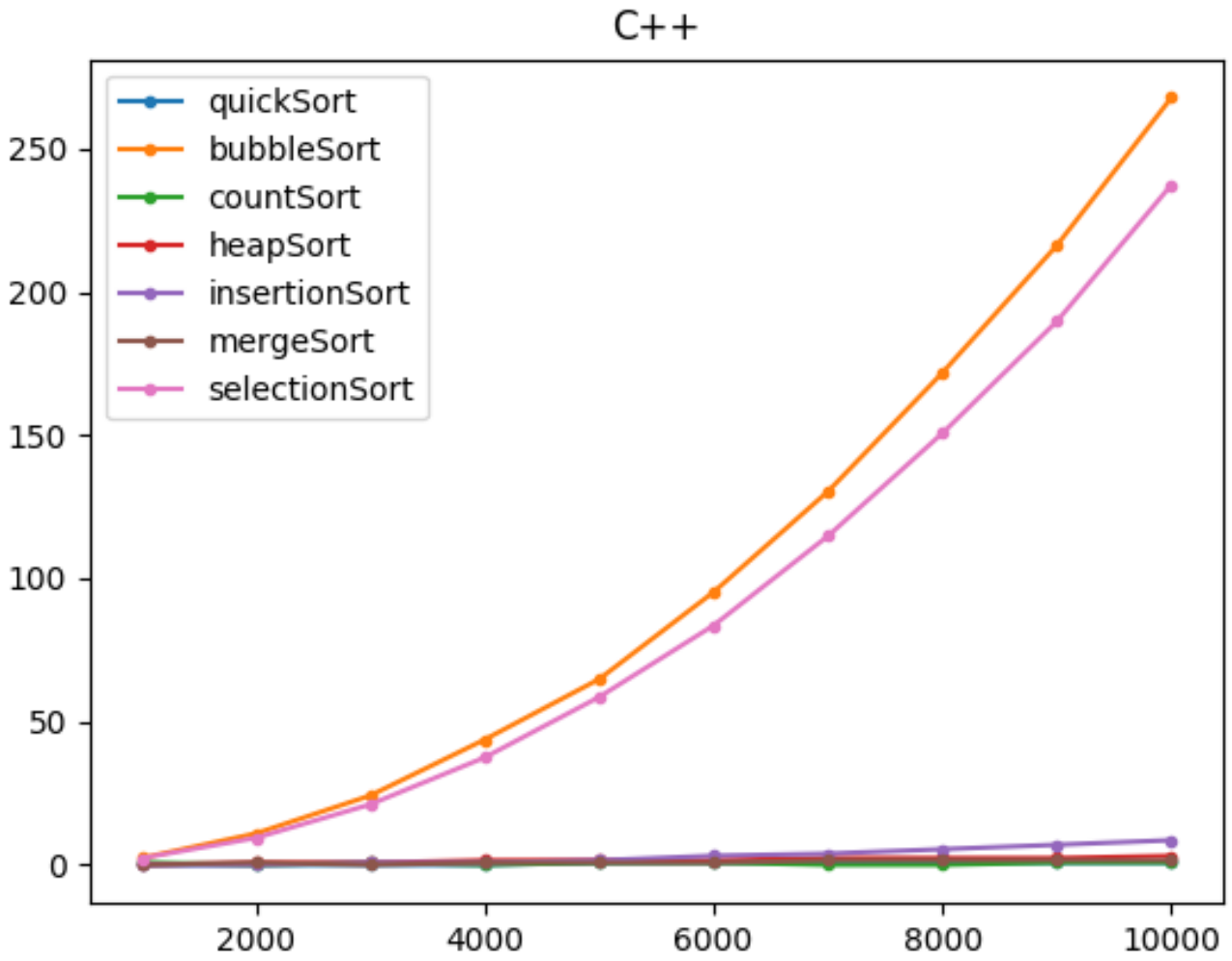
d) Gráfica comparativa del Merge Sort



Al ser un algoritmo con complejidad O(n log n) no es tan costoso y lo podemos verificar al observar la gráfica y corroborar con los tiempos que se demora para cada lenguaje, siendo Python siempre el que más se demora y como constante Java y C++ se mantienen a la par con alguna ventaja mínima de Java sobre C++. Cabe recalcar que nos volvemos a encontrar en la parte final de la gráfica con un escalón y esto se debe como ya se explicó más antes debido a la generación de datos, pero se puede apreciar que igual van por un intervalo no tan alejado en cada punto, esto para Java y C++.

8. **Gráficas comparativas de todos los algoritmos en C++**

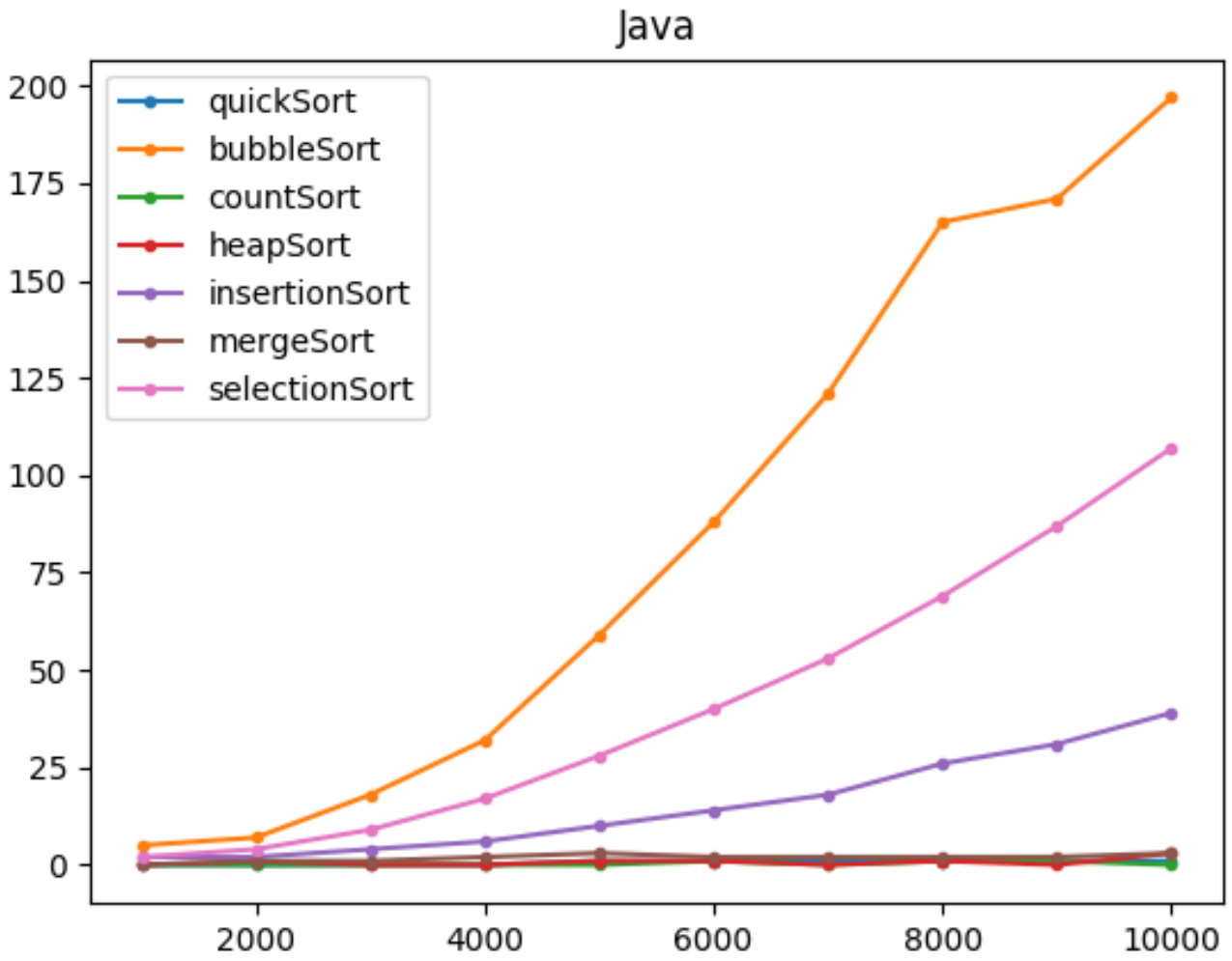
A continuación observamos como se comportan todos los algoritmos en C++:



Como podemos observar en la gráfica los algoritmos más costosos serían el Bubble Sort y el Selection Sort son los más cosotos al ser de complejidad n^2 , luego les sigue el Insertion que también tiene la misma complejidad pero no requiere de tantas operaciones para poder orndenarse, mientras que los dos anteriores sí.
Ya por debajo nos encontramos con el Heap sort, Quick Sort, Merge Sort que son de complejidad O(n log n) y por último que no se distingue mucho el Counting Sort que es de complejidad O(n+k).

9. **Gráficas comparativas de todos los algoritmos en Java**

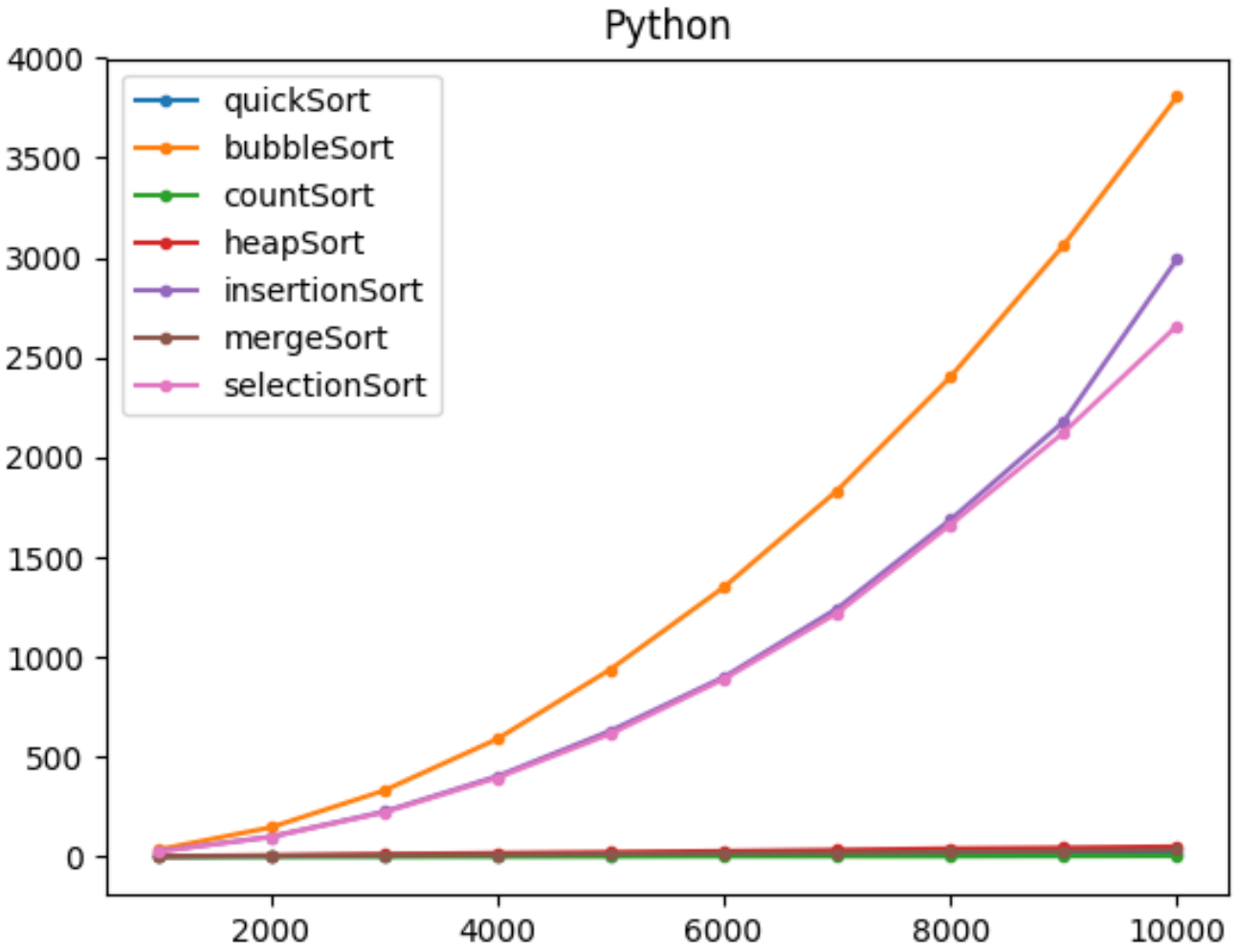
A continuación observamos como se comportan todos los algoritmos en Java:



Al igual que en C++ podemos observar que los algoritmos más costosos son el Bubble Sort y el Selection Sort al ser de complejidad n^2 , luego les sigue el Insertion que también tiene la misma complejidad pero no requiere de tantas operaciones para poder orndenarse, mientras que los dos anteriores sí.
Ya por debajo nos encontramos con el Heap sort, Quick Sort, Merge Sort que son de complejidad O(n log n) y por último que no se distingue mucho el Counting Sort que es de complejidad O(n+k). El Quick y el Merge al usar recursividad están en la misma línea pero si nos fijamos bien el Quick le lleva una ligera ventaja.

10. **Gráficas comparativas de todos los algoritmos en Python**

A continuación observamos como se comportan todos los algoritmos en Python:



Al igual que en C++ y Java podemos observar que los algoritmos más costosos son el Bubble Sort y el Selection Sort al ser de complejidad n^2 , luego les sigue el Insertion que también tiene la misma complejidad pero no requiere de tantas operaciones para poder orndenarse, mientras que los dos anteriores sí.
Vemos que para este lenguaje los tiempo son mucho mayores a los de las dos gráficas anteriores, en comparativa las gráficas de C++ y Java estaban hasta el tiempo 250 y 200 respectivamente pero en este lenguaje los

tiempos se incrementan hasta 4000, y esto se debe por lo que ya explicamos anteriormente, Python es muy costoso en sus bucles y en realizar sus operaciones, por ello emplea mucho más tiempo que los anteriores, esto también se da debido a que es un lenguaje de alto nivel en comparación con los otros les lleva ventaja en ese aspecto, por ello es mas costoso realizar operaciones con dicho lenguaje.
Pero si solo nos fijamo a nivel de los algoritmos nos damos cuenta que siguen cumpliendo ka regla de quienes son los más costosos y quienes son los menos costosos, situandose en última posición el Counting Sort debido a su complejidad también.

11. Conclusiones

a) Pude darme cuenta que la complejidad de los algoritmos es muy importante, ya que cuando realizamos el algortimo con una cantidad enorme de elementos puede llegar a demorar más dependiendo el algoritmo que sea.

b) El Counting Sort debe ser el más eficiente de todos los vistos, pero presenta limitaciones, como se explico en la parte de arriba, por ello es un algoritmo que nos servirá solo en ciertos casos.

c) Un buen uso de un algoritmo sería el Merge o Quick que ambos usan la técnica de divide y conquista, es más, lenguajes como Java lo utilizan para hacer sus funciones predeterminadas para ordenar.

d) Python es un lenguaje muy sencillo de usar, pero como nos hemos dado cuenta, al momento de realizar operaciones es muy costoso, principalmente por sus bucles.

e) Java y C++ son muy parecidos tanto en sus sintaxis como en el tiempo que se demora cada uno para realizar un determinado algoritmo.

f) Los algoritmos de ordenamiento son muy importantes, ya que en este mundo de la computación muchas veces se necesitan ordenar datos, ya sea nombres, apellidos, números y demás elementos que nos podamos imaginar y sin el uso de un buen algoritmo de ordenamiento podríamos poner en aprietos un sistema, ya que el tiempo que se demoraría en ordenar una cantidad de datos sería demasiada.

12. Referencias

https://www.geeksforgeeks.org/
https://www.ecured.cu/EcuRed
https://www.interviewbit.com/
https://www.cs.usfca.edu/~galles/visualization/